

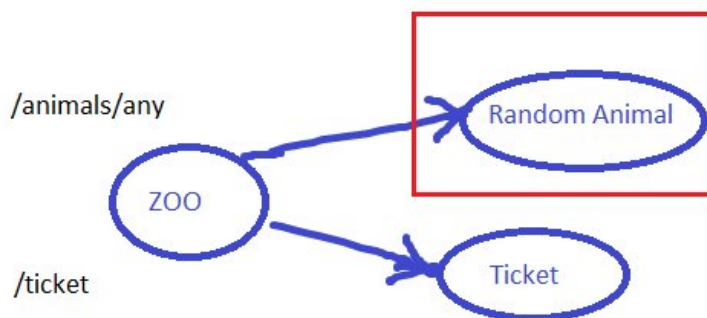
Отказоустойчивость микросервисов: шаблон Circuit Breaker

В этой статье речь пойдет об отказоустойчивости системы микросервисов и о том, как ее обеспечить. Мы рассмотрим шаблон проектирования Circuit Breaker и сравним его с шаблоном Bulkhead.

Также реализуем Circuit Breaker в Spring Cloud микросервисе с помощью библиотеки Resilience4j.

1 Система микросервисов

Итак, допустим у нас система из трех микросервисов:



Пользователь обращается в браузере к Zoo, а тот в свою очередь к Random Animal и Ticket.

Если пользователь ввел

```
1. /animals/any
```

то Zoo обращается к Random Animal (и возвращает Animal).

Если пользователь ввел

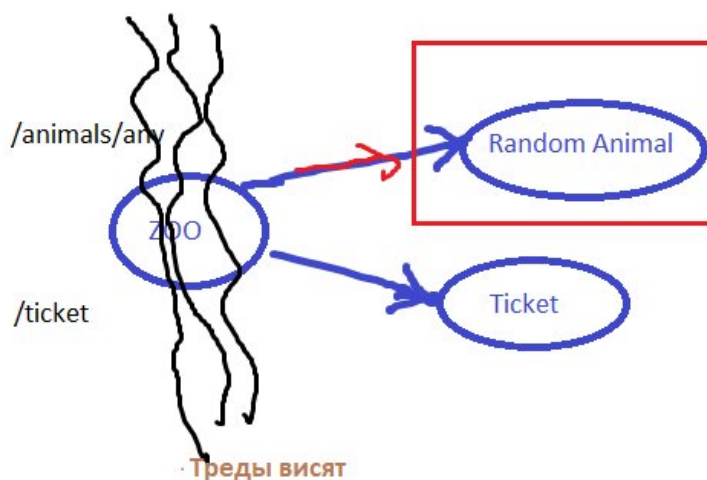
```
1. /ticket
```

то Zoo обращается к Ticket (и возвращает String — билет).

Допустим, микросервис Random Animal упал.

② Один микросервис упал — проблема для всей системы

Random Animal упал, значит по запросу */animals/any* животные больше не возвращаются. Но проблема в том, что теперь по запросу */ticket* билеты тоже могут перестать возвращаться.



Почему? Если число запросов

```
1. /animals/any
```

велико, а ответа от Random Animal приходится ждать долго, то запросы `/animals/any` быстро займут весь пул тредов, и на запросы `/ticket` свободных тредов не останется.

Ведь известно, что на каждый запрос выделяется отдельный тред. Запрос `/animals/any` заставляет тред веб-сервера Zoo висеть в ожидании ответа от Random Animal — то есть не дает треду быстро освободиться. А пул тредов на веб-сервере ограничен и един: он обслуживает и запросы `/animals/any`, и запросы `/ticket` и любые другие.

i

Если настроить `timeout` для *RestTemplate*, который обращается к Random Animal, то это может в каком-то случае решить проблему, а в каком-то не решить. Весь вопрос в том, освобождаются ли треды быстрее, чем приходят новые запросы.

Кстати, необязательно все запросы идут из браузера. В системе могут быть обычные микросервисы, обращающиеся к Zoo. И они тоже не смогут общаться с Zoo из-за того, что весь его пул тредов занят. А все из-за того, что упал единственный микросервис Random Animal.

Таким образом, один упавший элемент системы может завалить всю систему. И это не есть отказоустойчивость.

Ниже рассмотрим способы, как сделать систему более отказоустойчивой. К ним относятся:


- Запуск нескольких экземпляров Random Animal

- Circuit Breaker
- Bulkhead

i
1

Эти способы не исключают друг друга. Наоборот, лучше использовать их совместно.

3 Несколько экземпляров Random Animal

Вариант запуска нескольких экземпляров мы уже рассмотрели. В  про балансировку нагрузки мы запускали несколько экземпляров микросервиса. Да, второй живой экземпляр Random Animal спас бы ситуацию. Но допустим, они все упали.

Для такого случая есть совершенно другие подходы, которые мы и рассмотрим ниже — шаблоны Circuit Breaker и Bulkhead.

Но сначала о пара слов библиотеках, с помощью которых их можно реализовать.

4 Hystrix vs. Resilience4j

Есть две библиотеки — реализации шаблонов Circuit Breaker и Bulkhead:

- Hystrix
- Resilience4j

Hystrix уже не развивается и не совместим с последней версией Spring Cloud. Поэтому будем всё делать на Resilience4j.

5 Шаблон Circuit Breaker

Суть шаблона Circuit Breaker состоит в том, что если Zoo обнаруживает, что Random Animal работает плоховато (отвечает долго или выдает ошибки), то на некоторое время Zoo перестает обращаться к Random Animal, несмотря на внешние толчки. Как будто предохранитель срабатывает, схема открывается и все потоки перестают доходить.

Пользователь вводит в браузере этот злосчастный */animals/any*, но Zoo не обращается к Random Animal, а выдает некоторый стандартный ответ (fallback). И это окно ничегонеделания длится несколько секунд (в зависимости от настроек), в течение которых Zoo может сделать один или несколько пробных запросов к Random Animal. Если Zoo замечает, что Random Animal ожил, схема закрывается и нормальная работа возобновляется. Все происходит автоматически, программисту нужно только пометить опасные методы и задать настройки.

Опасный метод у нас — это обращение к Random Animal, то есть:

```
1. restTemplate.getForEntity("http://random-  
animal/random", Animal.class)
```

Его и будем пометить.

Параметры Circuit Breaker

Но у Circuit Breaker должны быть точные критерии, по которым он судит, что «все плохо» и переходит в открытое состояние, при котором указанный выше метод перестает вызываться. Эти критерии и задаются параметрами.

Параметры:

1. Circuit Breaker всегда оценивает последние вызовы опасного метода и по ним делает вывод. По умолчанию это количество вызовов (но может быть последние столько-то секунд). За значение количество вызовов отвечает параметр ***slidingWindowSize*** (значение по умолчанию=100).
2. Если хотя бы 50% из последних вызовов выдают ошибку (параметр ***failureRateThreshold*** =50), то схема открывается. И плохой метод больше не вызывается.
3. Есть также параметр ***slowCallDurationThreshold***=60000 мс: по этому параметру оценивается «медленность метода». Если некий процент вызовов медленнее этого параметра, то схема тоже закрывается. По умолчанию процент=100 (это параметр ***slowCallRateThreshold***).
4. Далее ***waitDurationInOpenState*** =6000 мс задает, сколько надо ждать перед тем, как снова проверить, не ожил ли микросервис.

Есть и другие , для всех из них существует значение по умолчанию, поэтому можно их не настраивать вообще. А можно все перезаписать. Подбор параметров делается на практике.

Реализация шаблона Circuit Breaker в Spring Cloud приложении

Для начала, добавим в микросервис Zoo зависимость:

```
1. <dependency>
2.     <groupId>org.springframework.cloud</groupId>
3.     <artifactId>spring-cloud-starter-
   circuitbreaker-resilience4j</artifactId>
4. </dependency>
```

Настройка параметров

Теперь настроим параметры:

```
1. @Configuration
```

```

2.     public class Resilience4JConfig {
3.         @Bean
4.         Customizer<Resilience4JCircuitBreakerFactory> def
5.         return factory -> factory.configureDefault(id
6.         Resilience4JConfigBuilder(id)
7.             .circuitBreakerConfig(
8.                 CircuitBreakerConfig.from(CircuitBreakerConfig.ofDefau
9.             )
10.            .build());
11.     }

```

Мы взяли параметры по умолчанию

CircuitBreakerConfig.ofDefaults(), перечисленные выше. А затем на основе их изменили параметр *.slidingWindowSize(3)*, чтобы судить о статусе микросервиса Random Animal не по 100 последним вызовам к нему, а по 3.

Вызовы *restTemplate* будут выводиться в консоль благодаря настройке в *application.properties*:

```

1. logging.level.org.springframework.web.client.RestTem
   = DEBUG

```

Таким образом, можно будет заметить, что после трех обновлений браузера *restTemplate* перестанет совершать вызов к упавшему микросервису. Это чтобы не обновлять браузер сто раз.

Как пометить метод

Внедряем бин *CircuitBreakerFactory* в класс *RandomAnimalClient*.

И затем вместо обычного вызова:

```

1.     public ResponseEntity<Animal> random() {
2.         return restTemplate.getForEntity("http://random-
3.         animal/random", Animal.class);

```

Делаем такой:

```

1.  public ResponseEntity<Animal> random() {
2.      LOGGER.debug("Sending request for animal {}");
3.
4.      return
5.      circuitBreakerFactory.create("randomAnimal").run(
6.          () ->
7.          restTemplate.getForEntity("http://random-
            animal/random", Animal.class),
            throwable -> fallbackRandom());
8.  }

```

здесь *fallbackRandom()* — метод, который выдает стандартный ответ в том случае, если микросервис Random Animal не работает, или когда обращение к нему вовсе не происходит. Он должен иметь ту же сигнатуру, что и реальный метод.

Таким образом, весь класс *RandomAnimalClient*:

```

1.  @Component
2.  public class RandomAnimalClient {
3.      private static final Logger LOGGER =
4.          LoggerFactory
5.              .getLogger(RandomAnimalClient.class);
6.
7.      private final RestTemplate restTemplate;
8.
9.      private final CircuitBreakerFactory
10.         circuitBreakerFactory;
11.
12.      public RandomAnimalClient(RestTemplate
13.          restTemplate, CircuitBreakerFactory
14.          circuitBreakerFactory) {
15.          this.restTemplate = restTemplate;
16.          this.circuitBreakerFactory =
17.              circuitBreakerFactory;
18.      }
19.
20.      public ResponseEntity<Animal> random() {
21.          LOGGER.debug("Sending request for animal
22.              {}");
23.
24.          return
25.          circuitBreakerFactory.create("randomAnimal").run(
26.              () ->
27.              restTemplate.getForEntity("http://random-
28.                  animal/random", Animal.class),
29.                  throwable -> fallbackRandom());
30.      }
31.
32.      public ResponseEntity<Animal> fallbackRandom()
33.      {

```



```
25.         return ResponseEntity.ok().body(new
Animal("no animal"));
26.     }
27. }
```

Проверка

Теперь попробуем запустить Eureka, Zoo. А Random Animal не запускать.

В браузере получим стандартный ответ:

```
> ↻ 🏠 ⓘ localhost:8081/animals/any

// 20210108201108
// http://localhost:8081/animals/any

{
  "name": "no animal"
}
```

Причем если обновлять браузер, то в консоли мы увидим обращение к `http://random-animal/random` только три раза:

```
1. 2021-01-08 20:23:40.359 INFO 20208 --- [nio-8081-
exec-1] o.s.web.servlet.DispatcherServlet :
Completed initialization in 0 ms
2. 2021-01-08 20:23:40.390 DEBUG 20208 --- [pool-1-
thread-1] o.s.web.client.RestTemplate :
HTTP GET http://random-animal/random
3. 2021-01-08 20:23:40.391 DEBUG 20208 --- [pool-1-
thread-1] o.s.web.client.RestTemplate :
Accept=[application/json, application/*+json]
4. 2021-01-08 20:23:44.217 DEBUG 20208 --- [pool-1-
thread-1] o.s.web.client.RestTemplate :
HTTP GET http://random-animal/random
5. 2021-01-08 20:23:44.218 DEBUG 20208 --- [pool-1-
thread-1] o.s.web.client.RestTemplate :
Accept=[application/json, application/*+json]
6. 2021-01-08 20:23:47.200 DEBUG 20208 --- [pool-1-
thread-1] o.s.web.client.RestTemplate :
HTTP GET http://random-animal/random
7. 2021-01-08 20:23:47.201 DEBUG 20208 --- [pool-1-
thread-1] o.s.web.client.RestTemplate :
Accept=[application/json, application/*+json]
```

При дальнейших обновления браузера *RestTemplate* ничего не делает, и этот же жестко закодированный ответ возвращается очень быстро, микросервис разгружен. Что и требовалось достигнуть.

Затем, если запустить отключенный микросервис *Random Animal*, то спустя небольшое количество секунд *Zoo* это замечает и работает нормально, выдавая реальный ответ.

Теперь перейдем к шаблону *Bulkhead* (реализовывать не будем, только описание).

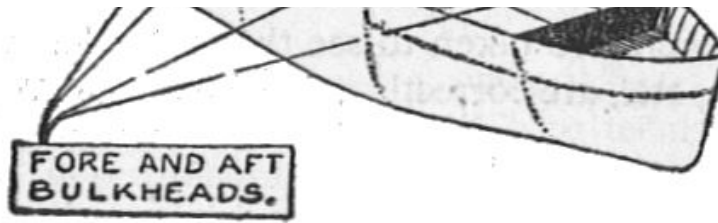
6 Шаблон Bulkhead

Шаблон *Bulkhead* подразумевает ограничение числа тредов на данный метод.

Как уже говорилось, пул тредов на веб-сервере един для всех запросов. Но с помощью библиотеки *Resilience4j* можно все же искусственно ограничить число тредов для метода. То есть разбить пул на несколько подпулов.

Мы задаем, что такой-то метод может занимать не более n тредов из пула, допустим 10 (а в веб-сервере *Tomcat* всего 100 тредов по умолчанию). Если приходит запрос, покушающийся на еще один тред, то возвращается ошибка.





Переборки (bulkheads)

Bulkhead переводится как «переборка», «отсек». Судно разделяют на отсеки, чтобы оно не затонуло. Если в одном отсеке пробоина, то вода только в нем. Также и пул тредов разделяется на подпулы-отсеки, тоже с целью не утопить весь веб-сервер. Отсюда название шаблона.

Bulkhead vs. Circuit Breaker

1. Таким образом, как и для Circuit Breaker, для BulkHead нужно тоже задать настройки (для BulkHead гораздо меньше).
2. Нужно пометить опасные методы — в Circuit Breaker они временно перестанут вызываться, а в BulkHead на них будут уходить не все треды.

7 Итоги

Мы рассмотрели и реализовали шаблон Circuit Breaker в Spring Cloud приложении. Исходный код доступен на [\[ссылка\]](#).

sysout / 08.01.2021 / Spring, Spring Cloud, Шаблоны

**Отказоустойчивость микросервисов:
шаблон Circuit Breaker: 1 комментарий**

Руслан

05.12.2023 в 12:53

Шикарная статья. Спасибо автору

Ответить

Добавить комментарий

Ваш адрес email не будет опубликован. Обязательные поля помечены *

КОММЕНТАРИЙ *

ИМЯ *

EMAIL *

САЙТ

ОТПРАВИТЬ КОММЕНТАРИЙ

НАЗАД

Spring Cloud Configuration Server

ДАЛЕЕ

Защита методов — аннотация `@PreAuthorize`

Прошу прощения: на комментарии временно не отвечаю.

СВЕЖИЕ КОММЕНТАРИИ

- sysout к записи [Отношение OneToMany в Hibernate и Spring](#)
- Сергей Акопов к записи [Отношение OneToMany в Hibernate и Spring](#)
- Дмитрий к записи [N+1 проблема в Hibernate](#)
- Дмитрий к записи [Spring Custom Login Form](#)
- Руслан к записи [Отказоустойчивость микросервисов: шаблон Circuit Breaker](#)
- Николай к записи [Spring Custom Login Form](#)
- Денис к записи [@Primary, @Qualifier и внедрение списка](#)

- Михаил к записи [Работа с IoC-контейнером в Spring](#)
- Арсен к записи [Как работает Flush в Hibernate](#)
- Сергей к записи [Spring Data JDBC: CrudRepository и Query methods](#)
- Yustas к записи [Пример приложения с JWT-токеном](#)
- sysout к записи [Пример приложения с JWT-токеном](#)
- Михаил к записи [Пример приложения с JWT-токеном](#)
- Андрей к записи [Преобразование Entity в DTO с помощью ModelMapper](#)

Core Java

Spring



Java Libraries

Git
